Lecture A2: TOY Programming

What We’ve Learned About TOY

TOY: what’s in it, how to use it.
- Von Neumann architecture.
- Box with switches and lights.

Data representation.
- Binary and hexadecimal.

TOY instructions.
- Instruction set architecture.

Sample TOY machine language programs.
- $1 + 2 + 3 + \ldots n$.
- LFBSR.
- Polynomial evaluation.

What We Do Today

Represent data other than positive integers.
- Negative numbers.

Manipulate addresses.
- Indexed addressing and "pointers."

Represent data structures.
- Arrays.

Implement functions.

Relate TOY, C, and "real computers".

Representing Negative Numbers (Two’s Complement)

<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>+32767</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>+4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>+2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>+1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>-2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>-3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>-4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>-32768</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

***
Two’s Complement Integers

Properties:
- Leading bit (bit 15) signifies sign.
- Negative integer \(-N\) represented by \(2^{16} - N\).
- Trick to compute \(-N\):
  1. Start with \(N\).
  2. Flip bits.
  3. Add 1.

Two’s Complement Arithmetic

Addition is carried out as if all integers were positive.
- It usually works:
  - Example:
    
    \[
    \begin{array}{c|c|c}
    \text{+32,767} & 0 & 11111111111111111111111111111111 \\
    \hline
    + & 0 & 00000000000000000000000000000000 \\
    \hline
    = & 2 & 00000000000000000000000000000000
    \end{array}
    \]

- Example:
  
  \[
  \begin{array}{c|c|c}
  \text{-32,767} & 1 & 00000000000000000000000000000001 \\
  \hline
  + & 0 & 00000000000000000000000000000000 \\
  \hline
  = & 2 & 00000000000000000000000000000001
  \end{array}
  \]

Two’s Complement Integers Properties

Nice properties:
- 0000000000000000 represents 0.
- -0 and +0 are the same.
- Addition is easy (see next slide).

Not-so-nice properties:
- Can represent one more negative integer than positive integer
  \((-32,768 = -2^{15} \text{ but not } 32,768 = 2^{15})\).

Alternatives other than two’s complement exist.
- Many C compilers use two’s complement.
- But not all, so do not assume they do.
- Unsafe C code to test if \(a\) is odd: \(\text{if } (a \& 1)\)
Representing Other Primitive Data Types

Big integers.
- Can use "multiple precision."
- Use two 16-bit words per integer.

Real numbers.
- Can use "floating point" (like scientific notation).
- Double word for extra precision.

Character strings.
- Can use ASCII code (8 bits / character).
- Can pack two characters into one 16-bit word.

Indexed Addressing

Static addressing.
- So far, all load/store addresses hardwired inside instruction.
- Ex. 9234: R2 ← mem[34]
- Need more flexibility to implement arrays, functions, etc.

Indexed (dynamic) addressing.
- Want to be able to make memory index a variable, instead of hardwiring '34'.

Solution.
- Put memory address in register. (C "pointer")
- Use CONTENTS of register as address.
- Augment instruction format to use address register.

Review: Format 2 Instructions

Register-memory / register-immediate.
- Bits 12-15 encode opcode.
- Bits 8-11 encode destination register.
- Bits 0-7 encode memory address or arithmetic constant.

Ex: 9234 means
- Load contents of memory location 34₁₆ into register R2.
- R2 ← mem[34]

Indexed Addressing

Bits 11 signifies "indexed addressing."
- If Bit 11 is 0 then Format 2 as usual.
- If Bit 11 is 1 then replace addr by R1 + R2
- 9234 means R2 ← mem[34]
- 9A34 means R2 ← mem[R3 + R4]
Why "Stealing" Bit 11 is OK

Bits 11 signifies "indexed addressing."
- We only have 8 registers.
- Only 3 bits (8-10) needed to distinguish among 8 values.
- Can "steal" bit 11.

Could we do the same for Format 1 instructions?

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>opcode</th>
<th>dest</th>
<th>addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>9_{16}</td>
<td>2_{16}</td>
<td>34_{16}</td>
</tr>
</tbody>
</table>

Sample C Program: Array

Goal: put Fibonacci numbers into array a[].
- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

```c
int main(void) {
    int n, i, j, k, d[16];
    n = 10;
    d[0] = 1; d[1] = 1;
    i = 0; j = 1; k = 2;
    do {
        d[k] = d[i] + d[j];
        i++; j++; k++;
    } while (--n > 0)
    return 0;
}
```

Sample TOY Program 3: Array

```toy
use indexed addressing three times

fibonacci.toy

10: B10A R1 <- 000A
11: B001 R0 <- 0001
12: B2D0 R2 <- 00D0   a
13: A0D0 mem[D0] <- 1  a[0] = 1
14: A0D1 mem[D1] <- 1  a[1] = 1
15: B300 R3 <- 0     i = 0
16: B401 R4 <- 1     j = 1
17: B502 R5 <- 2     k = 2
18: 9E23 R6 <- mem[R2 + R3] a[i]
19: 9F24 R7 <- mem[R2 + R4] a[j]
1A: 1667 R6 <- R6 + R7
1B: AE25 mem[R2 + R5] <- R6 a[k]
1C: 1330 R3++     i++
1D: 1440 R4++     j++
1E: 1550 R5++     k++
1F: 7118 to 18 if --R1 > 0
```

Food for Thought

What happens if we change B10A to B1AA?

```
mystery.toy

10: B10A R1 <- 000A
11: B001 R0 <- 0001
12: B2D0 R2 <- 00D0
13: A0D0 mem[D0] <- 1
14: A0D1 mem[D1] <- 1
15: B300 R3 <- 0
16: B401 R4 <- 1
17: B502 R5 <- 2
18: 9E23 R6 <- mem[R2 + R3]
19: 9F24 R7 <- mem[R2 + R4]
1A: 1667 R6 <- R6 + R7
1B: AE25 mem[R2 + R5] <- R6 a[k]
1C: 1330 R3++     i++
1D: 1440 R4++     j++
1E: 1550 R5++     k++
1F: 7118 to 18 if --R1 > 0
```
Branches and Loops

Press GO, TOY machine either:
- Executes some instructions and halts.
- Gets caught in an infinite loop.

Infinite loop.
- Puzzles and/or panics programmers. Why doesn’t compiler detect and tell me?
- Control structures (while, for) help manage control flow and avoid looping.
- Can always top machine by pulling plug! (Ctrl-c)

Function Calls

Functions can be used and written by different people.

Issues:
- How to pass parameter values?
- How to know where to return? (may have multiple calls)

One solution: adhere to CALLING conventions.
- Agreement between function and calling program on where to store parameters and return address.
- Assume parameter value(s) in specific register(s).
- Assume return value(s) in specific register(s).
- Save return address (jump-and-link).
- Use indexed jump to return.

TOY Program 4: Function Call

Goal: create function to compute \( a^b \).

Calling convention. Store:
- 0 in R0
- a in R1
- b in R2
- return address in R4
- result in R3

How to compute \( a^b \)?
- Set R3 = 1.
- Loop b times.
  - multiply R3 by a each time

TOY Program 4: Function Call

Client program to compute \( x^4 + y^5 \). Assume
- x in memory location D0
- y in memory location D1
How To Build a TOY Machine

Hardware.
- See Lecture A3-A5.

Simulate in software.
- Write a program to "simulate" the behavior of the TOY machine.
- Java TOY simulator.
- C TOY simulator.

```c
int main(void) {
  short int inst, R[8], mem[256];
  unsigned char pc = 0X10;
  int i, op, addr, r0, r1, r2, c;
  for (i = 0; i < 256; i++)
    mem[i] = 0;
  while (scanf("%hX%hX", &i, &inst) != EOF)
    mem[i] = inst;
  do {
    inst = mem[pc++];
    op   = (inst >> 12) & 15;
    r0   = (inst >>  8) &  7;
    r1   = (inst >>  4) &  7;
    r2   = (inst >>  0) &  7;
    addr = (inst >>  0) & 255;
    if ((inst >> 11) & 1)
      addr = (R[r1] + R[r2]) & 255;
    . . .
  } while (op != 0);
  return 0;
}
```

TOY SIMULATOR: toy.c
- initialize memory to 0
- read program
- fetch and increment
- execute

Shifting and Masking

Extract destination register.
- Given 16 bit integer in C, isolate bits 8-10.
- Use bit operations in C.

```
inst = B204_{16} = 45572_{10}

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(inst >> 8)

<table>
<thead>
<tr>
<th>7</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

7

<table>
<thead>
<tr>
<th>7</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

(inst >> 8) & 7

<table>
<thead>
<tr>
<th>7</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

R6
```

switch (op) {
  case 0: break;
  case 1: R[r0] = R[r1] + R[r2]; break;
  case 2: R[r0] = R[r1] - R[r2]; break;
  case 3: R[r0] = R[r1] * R[r2]; break;
  case 4: printf("%04X\n", R[r0]); break;
  case 5: pc = addr; break;
  case 6: if (R[r0] > 0) pc = addr; break;
  case 7: if (--R[r0]) pc = addr; break;
  case 8: R[r0] = pc; pc = addr; break;
  case 9: R[r0] = mem[addr]; break;
  case 10: mem[addr] = R[r0]; break;
  case 11: R[r0] = addr; break;
  case 12: R[r0] = R[r1] ^ R[r2]; break;
  case 13: R[r0] = R[r1] & R[r2]; break;
  case 14: R[r0] = R[r0] >> addr; break;
  case 15: R[r0] = R[r0] << addr; break;
}
Simulation

Consequences of simulation.
- Test out new machine (or microprocessor) using simulator.
  - cheaper and faster than building actual machine
- Easy to add other functions to simulator.
  - trace, single-step, breakpoint debugging
  - simulator more powerful than TOY itself
- Reuse software for old machines.

Ancient programs still running on modern computers.
- Ticketron.
- Lode Runner on Apple Ile.

C and TOY

Correspondence between C constructs and TOY mechanisms.

<table>
<thead>
<tr>
<th>C</th>
<th>TOY</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment</td>
<td>load, store</td>
</tr>
<tr>
<td>arithmetic expressions</td>
<td>add, multiply, subtract</td>
</tr>
<tr>
<td>logical expressions</td>
<td>xor, and, shifts</td>
</tr>
<tr>
<td>loops (for, while)</td>
<td>jump and count</td>
</tr>
<tr>
<td>branches (if-else, switch)</td>
<td>jump if positive, jump</td>
</tr>
<tr>
<td>arrays, linked lists</td>
<td>indexed addressing</td>
</tr>
<tr>
<td>function call</td>
<td>jump and link</td>
</tr>
<tr>
<td>recursion</td>
<td>implement stack with arrays</td>
</tr>
<tr>
<td>whitespace</td>
<td>D000</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
</tr>
</tbody>
</table>

Bootstrapping

Translate TOY program into C?

Translate C program to TOY?

Translate TOY simulator into TOY?

Bootstrapping.
- Build "first" machine.
- Implement simulator of itself.
- Modify simulator to try new designs.
  (still going on!)